Multi-Rate Processing

Introduction

This project has three objectives

1. To develop experience performing digital signal processing
2. To solidify understanding of multi-rate processing, interpolation and decimation, and poly-phase filters
3. To design an appropriate digital filter

Filter Design

We were required to design a filter with a small ripple in the pass-band (< .001 dB), and a sharp peak side-lobe level (<-80dB). We were interpolating by a factor of 3/2, thus the more restrictive LPF required was one with a normalized frequency of 1/3 (0.33). To achieve this in a non-ideal scenario I gave myself some wiggle room on the pass (0.31) and stop (0.35) bands. I used the default FIR filter design method provided by the Matlab tool (Equiripple).

To determine the order I had the tool generate a minimal filter, which used 248 taps. For convenience in coding I then manually designed a filter with the next length divisible by 6 (thus it later divided evenly into the different polyphaser filters), 252. In setting the number of taps I wasn't able to be as specific about the pass-band ripple or the stop-band peak, but the filter didn't appear to change too much. Upon inspection the side-lobe peak was about 85 dB, and the pass-band ripple about $5x10^{-4}$ dB (.0005 dB), thus well within the requirements. Note that for some reason Matlab defines the order as the number of taps less one.

See the Figures 1-6 for details of use of the Matlab filter design tool:

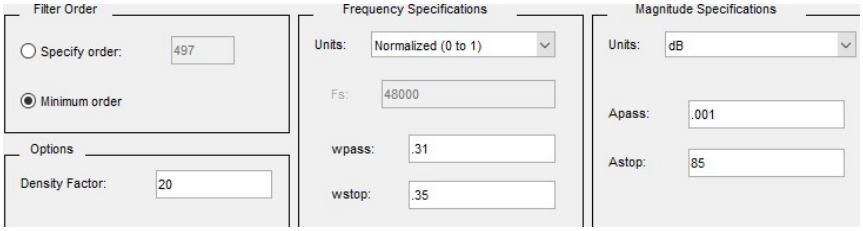

Figure 1 - Minimal Order Filter Design Specifications

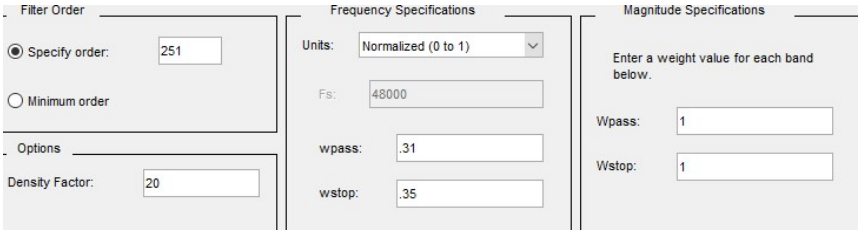

Figure 2 - Modulus 6 Order Filter Design Specifications

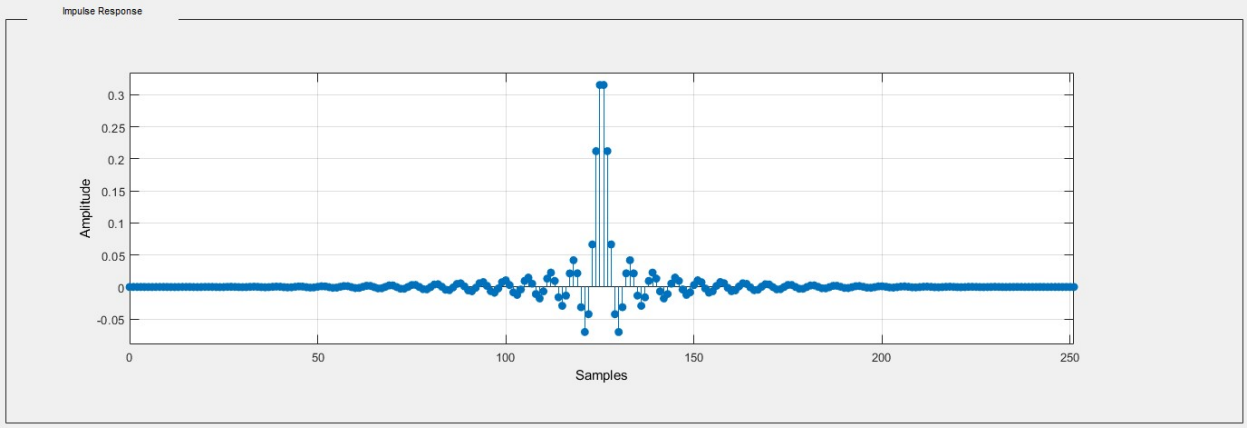

Figure 3 - Filter Impulse Response

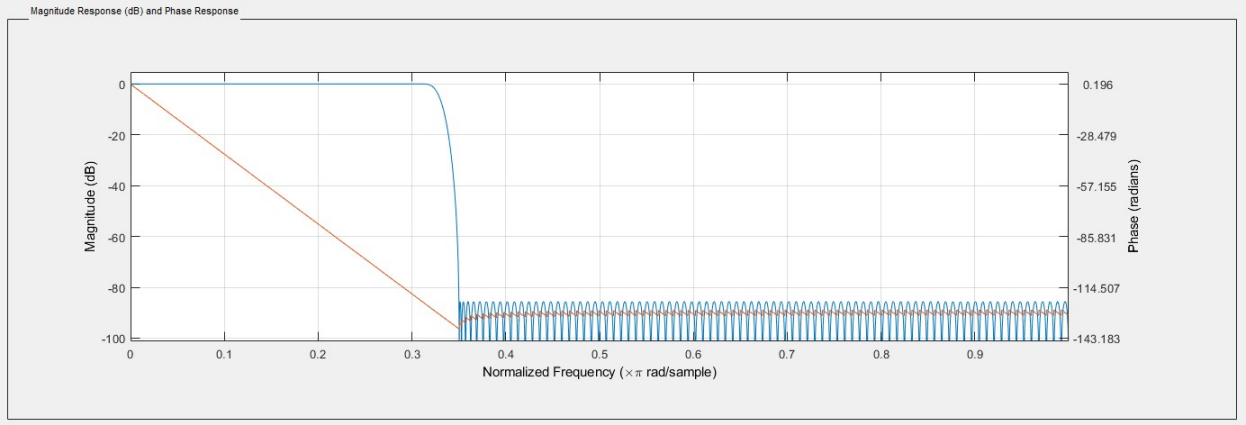

Figure 4 - Filter Transfer Function

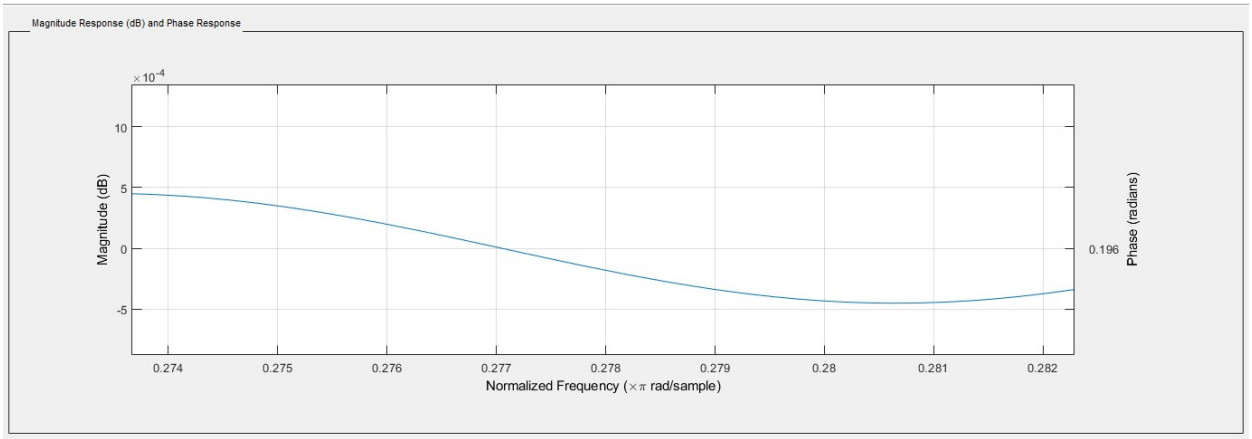

Figure 5 - Filter Passband Ripple Zoomed In

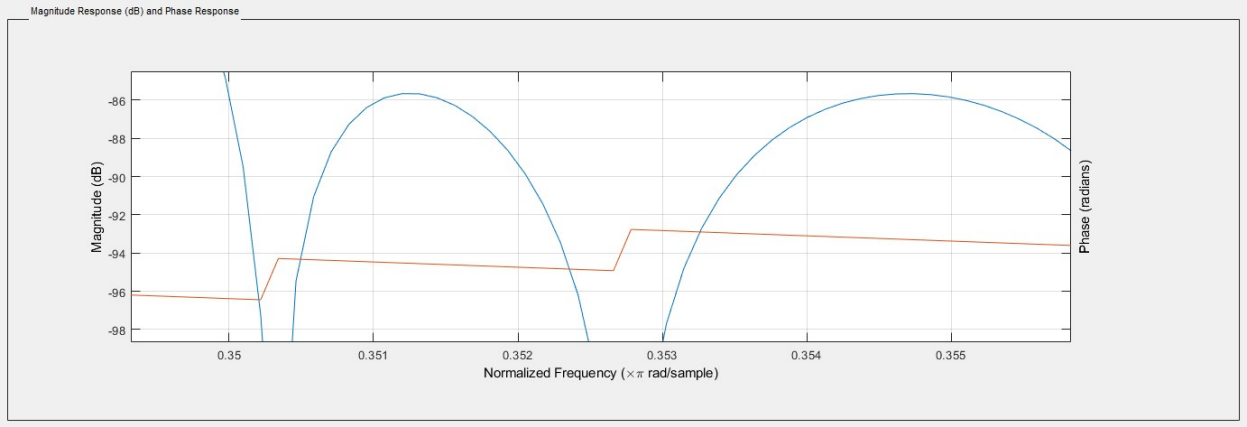

Figure 6 - Filter Side-Lobe Peak Zoomed In

Finally to maintain consistent amplitude of the original signal as a result of the interpolation process the filter was scaled to have a gain of 3.

Polyphase Filter Design

Much of the effort of designing the polyphase filter was derived in class lectures. In class we derived a 2/3 decimator, so it was a simple restructure to make a 3/2 interpolator.

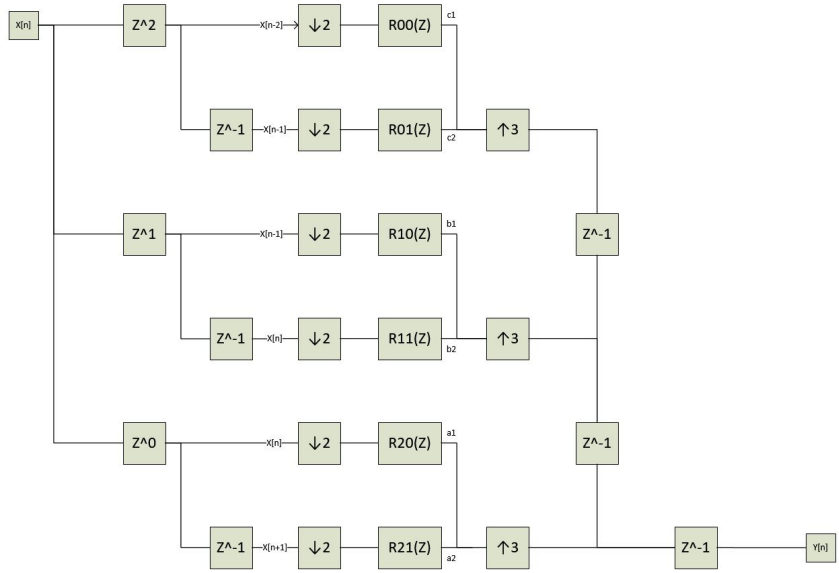

Figure 7 - Polyphase flow (Image made by peer Taylor Peterson, without whom this report would have a scanned hand-drawing instead due to lack of experience with Visio and lack of time)

In class we derived the number of MPUs to be $\frac{N+1}{M}$ and the number of APUs $\frac{N+1-L}{M}$. N is the side of our filter, 252. Thus we conclude this system would require 127 MPUs and 125 APUs. Of note (from the assignment description), a 3/2 interpolator given an input of 11.025 kHz would output a signal of 16.538 kHz.

In the code you will see me reference R_{ud}. The indices u and d correspond with the graph above. $R_{ud}[n]$ is defined such that the coefficients of h[n] (as defined in the previous section) are allocated as follows:

{ $R_{00}[0]$, $R_{01}[0]$, $R_{10}[0]$, $R_{11}[0]$, $R_{20}[0]$, $R_{21}[0]$, $R_{00}[1]$, $R_{01}[1]$, $R_{10}[1]$, $R_{11}[1]$, $R_{20}[1]$, $R_{21}[1]$, $R_{00}[2]$, … }

Testing the filters with sinusoids

In our lab we tested the digital resampling using 3 sinusoidal frequencies, 1/16, 1/8, and 1/4. These were generated in matlab with N = 100,000 so as to be well larger than the size of our filter. See Figures 8-10 for the results of resampling these sinusoids with the two approaches. Note that the "pol" suffix is used to indicate the results of the polyphase approach and the "dig" suffix is used to indicate the results of the direct approach. Also note that the plots were generated using the signal analyzer matlab tool.

f_0	f_{direct}	$f_{polyphase}$
1/16	3/32	3/32
1/8	3/16	3/16
1/4	3/8	3/8

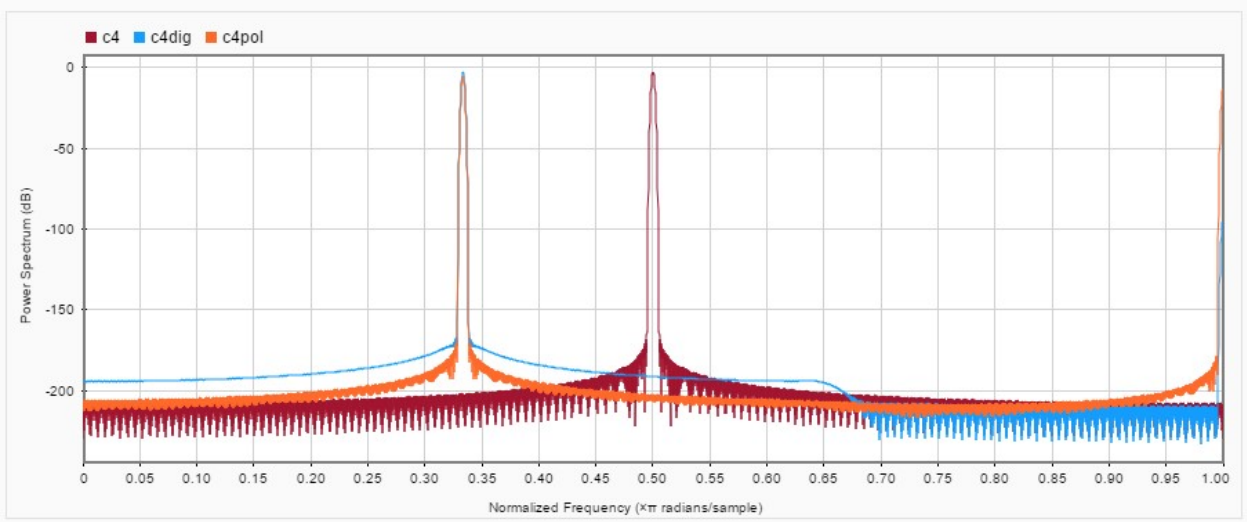

Figure 8 - A 1/4 sinusoid experiencing interpolation

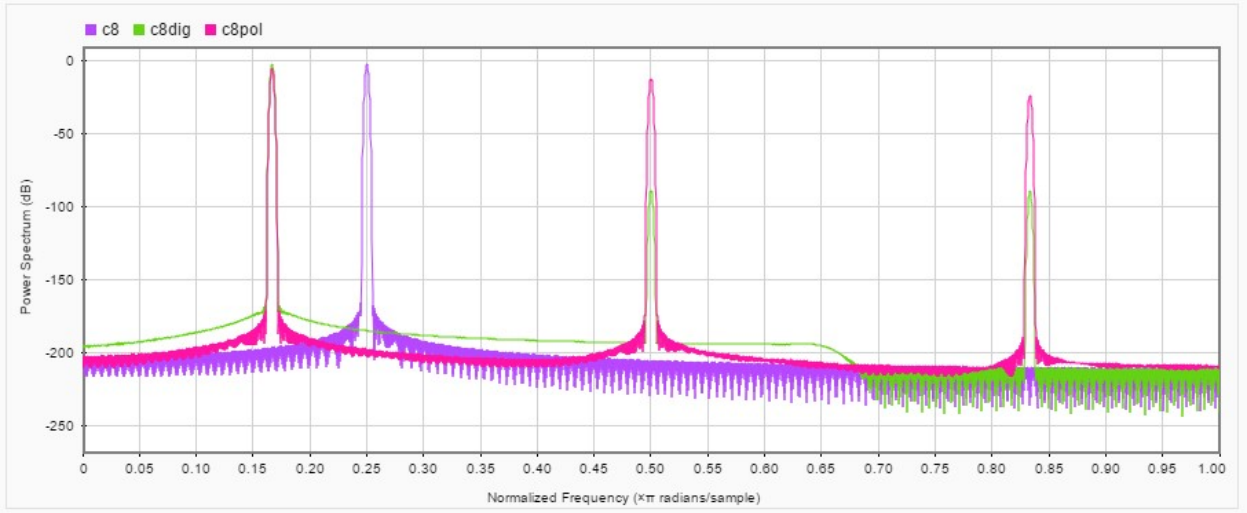

Figure 9 - A 1/8 sinusoid experiencing interpolation

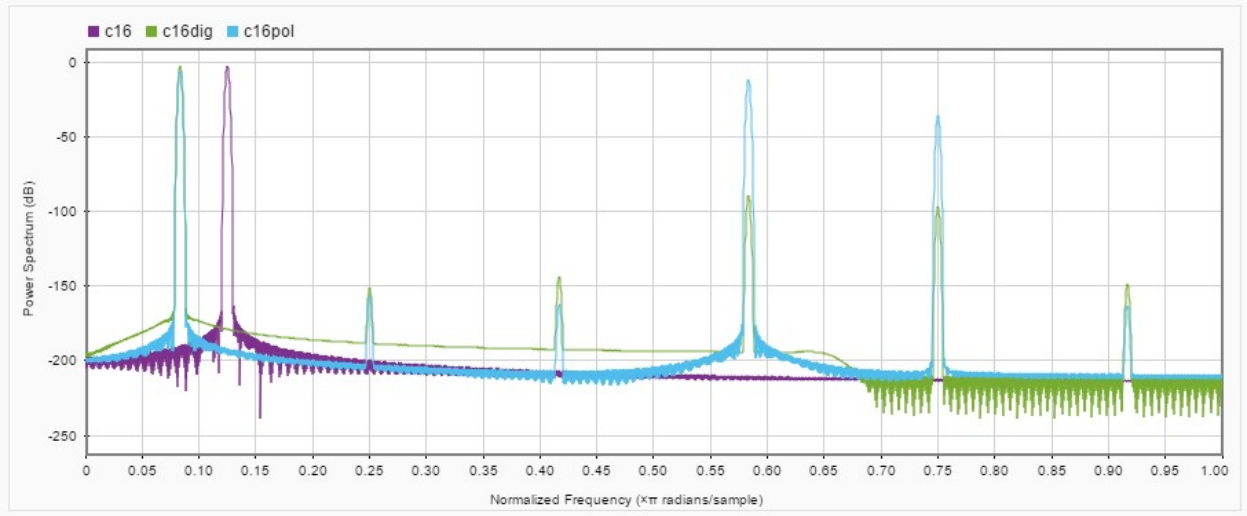

Figure 10 - A 1/16 sinusoid experiencing interpolation

Conclusion

To be honest I didn't expect spectral islands to show (see Figures 9-10) as a result of resampling. This suggests that either there is a problem with the filter design or the convolution as I have implemented it but I have not been able to pin down the exact problem. The re-samplers were also run on the Ghostbusters clip provided in class; the straight-forward approach produces an audio file that is not discernibly different from the original. However the polyphase filter approach has a discernable "tin-like" sound to it.

Despite difficulties in the implementation, it is understood how the polyphase approach can be advantageous. Though I did my best (in a naïve way) to maximize computational usage of the resampling in the direct approach, there are still computed values that are later discarded (which could perhaps be optimized with a bit more effort). Even so, the polyphase approach is still able to reduce the number of multiplication operations done to produce an output sample, and in a real-time environment less time is used idling and the workload is more balanced.

One final note: you may see my code load the entire input, where a requirement of the lab is that this assignment be viable for arbitrary length signals. Despite appearances, you will notice the resampler classes use a feed function which uses one input signal value at a time. Thus this code is perfectly viable in real-time scenarios or for indefinite length signals; it simply made sense in this scenario to load the whole file (no sense in wasting time on IO).

Appendix: C++ and Matlab Code

signal.hpp

```
#pragma once
#include <functional>
#include <string>
#include "types.h"

template <typename T>
class Signal {
private:
	inline void clean() {
		delete[] signal;
		signal = nullptr;
		length = 0;
	}
	inline void copy(const Signal<T>& rhs) {
		length = rhs.length;
		signal = new T[length];
		memcpy(signal, rhs.signal, sizeof(T) * length);
	}

protected:
	int length;
	T* signal;

	Signal() {
		length = 0;
		signal = nullptr;
	}

public:
	friend int OpenBin(std::string, Signal<float>**);
	friend int SaveBin(std::string, Signal<float>*);

	typedef std::function<void(int, T)> accessor;
	typedef std::function<T(int, T)> mutator;
	typedef std::function<T(int)> setter;

	inline void each(accessor f) const {
		for (int n = 0; n < length; ++n) {
			f(n, signal[n]);
		}
	}
	inline void each(mutator f) {
		for (int n = 0; n < length; ++n) {
			signal[n] = f(n, signal[n]);
		}
	}
	Signal(int l, setter f) {
		length = l;
		signal = new T[length];
		for (int n = 0; n < length; ++n) {
			signal[n] = f(n);
		}
	}

	Signal(int l) {
		length = l;
		signal = new T[length]();
	}
	Signal(int l, T v) {
		length = l;
		signal = new T[length];
```

```
                for (int n = 0; n < length; ++n) {
                        signal[n++] = v;
                }
        }
        Signal(int l, T* sig) {
                length = l;
                signal = new T[length];
                memcpy(signal, sig, sizeof(T) * length);
        }
        Signal<T>& operator=(const Signal<T>& rhs) { clean(); copy(rhs); return *this; }
        Signal(const Signal<T>& rhs) { copy(rhs); }
        ~Signal() { clean(); }

        inline const int N() const { return length; }
        inline const int ConvTailN() const { return length / 2; }

        inline T Get(int n) const {
                if (n < 0 || n >= length || signal == nullptr) { return 0; }
                return signal[n];
        }
        inline void Set(int n, T val) {
                if (n < 0 || n >= length || signal == nullptr) { return; }
                signal[n] = val;
        }
};

#define ERROR_BIN_FILE          (1 << 0)
#define ERROR_PGM_HEADER        (1 << 1)
#define ERROR_PGM_MAXSIZE       (1 << 2)

int OpenBin(std::string, Signal<float>**);
int SaveBin(std::string, Signal<float>*);
```

signal.cpp

```
#include "signal.hpp"
#include <fstream>


int OpenBin(std::string file, Signal<float>** iin) {
        std::fstream fin(file, std::ios::binary | std::ios::in);
        if (!fin) {
                return ERROR_BIN_FILE;
        }

        int N;
        fin.read((char*)&N, sizeof(int));
        Signal<float>* in = new Signal<float>(N);
        fin.read((char*)(in->signal), sizeof(float) * N);
        fin.close();
        *iin = in;
        return 0;
}

int SaveBin(std::string file, Signal<float>* out) {
        std::fstream fout(file, std::ios::binary | std::ios::out);
        fout.write((char*)out->length, sizeof(int));
        fout.write((char*)out->signal, out->length * sizeof(float));
        fout.close();
        return 0;
}
```

main.cpp

```
#include <iostream>
#include <fstream>
#include "signal.hpp"

class DigiResampler {
private:
	int U, D, N;
	Signal<float>* h;
	float* buff;

	std::ofstream* fout;

	int buff_i, y_i;

public:
	DigiResampler(int up, int down, Signal<float>* filter, std::ofstream* file) {
		U = up;
		D = down;
		h = filter;
		N = h->N();
		buff = new float[N]();
		buff_i = y_i = 0;
		fout = file;
	}
	~DigiResampler() {
		delete[] buff;
	}

	DigiResampler(const DigiResampler& rhs) = delete;
	DigiResampler& operator=(DigiResampler const& rhs) = delete;

	void feed(float xn) {
		// Convolve xn
		for (int i = 0; i < N; ++i) {
			buff[(buff_i + i) % N] += xn * h->Get(i);
		}

		// Upsample (buff_i increments to simulate inserting 0's)
		int y_f = buff_i + U;
		buff_i = y_f % N;

		// Downsample
		while (y_i < y_f) {
			// Output buff[y_i % N]
			float y_o = buff[y_i % N];
			fout->write((char*)(&y_o), sizeof(float));

			for (int d = 0; d < D; ++d) {
				buff[(y_i + d) % N] = 0;
			}

			// Discard D-1 values
			y_i += D;
		}
		// Done here so overflow doesn't prevent us from knowing when to stop
		y_i %= N;
	}
};

class PolyResampler {
private:
	struct PolyFilter {
		float* Filter;
		float* Buffer;
	};
	PolyFilter* R;
	int U, D, Rn;
```

```
	int Ri;

	std::ofstream* fout;

	inline PolyFilter* GetFilter(int u, int d) {
		if (u < 0 || d < 0 || u < U || d < D) {
			return nullptr;
		}
		return &R[u*D + d];
	}

	PolyResampler(const PolyResampler& rhs) = delete;
	PolyResampler& operator=(PolyResampler const& rhs) = delete;

public:
	~PolyResampler() {
		for (int u = 0; u < U; ++u) {
			for (int d = 0; d < D; ++d) {
				PolyFilter* Rud = &R[u*D + d];
				delete[] Rud->Buffer;
				delete[] Rud->Filter;
			}
		}
		delete[] R;
	}

	// In this case, I believe (naive approach) it will be more cache efficient to have each Rud
filter be contiguous
	PolyResampler(int up, int down, Signal<float>* filter, std::ofstream* file) {
		fout = file;
		U = up;
		D = down;
		Rn = filter->N() / 6;
		R = new PolyFilter[U * D];
		for (int u = 0; u < U; ++u) {
			for (int d = 0; d < D; ++d) {
				PolyFilter* Rud = &R[u*D + d];
				Rud->Buffer = new float[Rn]();
				Rud->Filter = new float[Rn];

				// Generate the smaller filters
				for (int n = 0; n < Rn; ++n) {
					Rud->Filter[n] = filter->Get((n * U * D) + (u * D) + d);
				}
			}
		}
		Ri = 0;
	}

	void feed(float xn) {
		int d = Ri % D;

		// Feed xn to all R-filters at the d-offset
		for (int u = 0; u < U; ++u) {
			PolyFilter* Rud = &R[u*D + d];

			// Convolve xn
			for (int n = 0; n < Rn; ++n) {
				Rud->Buffer[(Ri + n) % Rn] += xn * Rud->Filter[n];
			}
		}

		int nextRi = (Ri + 1) % Rn;
		d = nextRi % D;

		// After x[0], x[1], ..., x[D] has been feed through
		if (d == 0) {
			// Output y[0], y[1], ..., y[U]
			for (int u = 0; u < U; ++u) {
```

```
                                float Run = 0;
                                for (int d = 0; d < D; ++d) {
                                        PolyFilter* Rud = &R[u*D + d];
                                        Run += Rud->Buffer[Ri];
                                        Rud->Buffer[Ri] = 0;
                                }
                                fout->write((char*)(&Run), sizeof(float));
                        }
                }

                Ri = nextRi;
        }

};

#define INTERP_UP       3
#define INTERP_DOWN     2

int main() {
        int err;
        Signal<float>* h = nullptr;

        err = OpenBin("lpf_scaled.bin", &h);
        if (err != 0) {
                return err;
        }
        if (h == nullptr) {
                std::cout << "lpf not parsed correctly" << std::endl;
                return -1;
        }

        int N;
        float* x;

        // Ghostbusters
        std::ifstream fin("ghostbustersray.bin", std::ios::binary | std::ios::in);
        fin.read((char*)&N, sizeof(int));
        x = new float[N];
        fin.read((char*)x, sizeof(float) * N);
        fin.close();

        std::ofstream fDigOut("digInterp.bin", std::ios::binary | std::ios::out);
        std::ofstream fPolOut("polInterp.bin", std::ios::binary | std::ios::out);
        DigiResampler DigInterp(INTERP_UP, INTERP_DOWN, h, &fDigOut);
        PolyResampler PolInterp(INTERP_UP, INTERP_DOWN, h, &fPolOut);

        for (int i = 0; i < N; ++i) {
                DigInterp.feed(x[i]);
                PolInterp.feed(x[i]);
        }
        fDigOut.close();
        fPolOut.close();
        delete x;

        // 1/16 freq cosine
        fin.open("c16.bin", std::ios::binary | std::ios::in);
        fin.read((char*)&N, sizeof(int));
        x = new float[N];
        fin.read((char*)x, sizeof(float) * N);
        fin.close();

        fDigOut.open("digc16.bin", std::ios::binary | std::ios::out);
        fPolOut.open("polc16.bin", std::ios::binary | std::ios::out);
        DigiResampler DigC16(INTERP_UP, INTERP_DOWN, h, &fDigOut);
        PolyResampler PolC16(INTERP_UP, INTERP_DOWN, h, &fPolOut);
        for (int i = 0; i < N; ++i) {
                DigC16.feed(x[i]);
                PolC16.feed(x[i]);
        }
```

```
        fDigOut.close();
        fPolOut.close();
        delete x;

        // 1/8 freq cosine
        fin.open("c8.bin", std::ios::binary | std::ios::in);
        fin.read((char*)&N, sizeof(int));
        x = new float[N];
        fin.read((char*)x, sizeof(float) * N);
        fin.close();

        fDigOut.open("digc8.bin", std::ios::binary | std::ios::out);
        fPolOut.open("polc8.bin", std::ios::binary | std::ios::out);
        DigiResampler DigC8(INTERP_UP, INTERP_DOWN, h, &fDigOut);
        PolyResampler PolC8(INTERP_UP, INTERP_DOWN, h, &fPolOut);
        for (int i = 0; i < N; ++i) {
                DigC8.feed(x[i]);
                PolC8.feed(x[i]);
        }
        fDigOut.close();
        fPolOut.close();
        delete x;

        // 1/4 freq cosine
        fin.open("c4.bin", std::ios::binary | std::ios::in);
        fin.read((char*)&N, sizeof(int));
        x = new float[N];
        fin.read((char*)x, sizeof(float) * N);
        fin.close();

        fDigOut.open("digc4.bin", std::ios::binary | std::ios::out);
        fPolOut.open("polc4.bin", std::ios::binary | std::ios::out);
        DigiResampler DigC4(INTERP_UP, INTERP_DOWN, h, &fDigOut);
        PolyResampler PolC4(INTERP_UP, INTERP_DOWN, h, &fPolOut);
        for (int i = 0; i < N; ++i) {
                DigC4.feed(x[i]);
                PolC4.feed(x[i]);
        }
        fDigOut.close();
        fPolOut.close();
        delete x;

        delete h;
        return 0;
}
```

matlab code

```
[x, Fs] = audioread('ghostbustersray.wav');
fid = fopen('ghostbustersray.bin', 'wb');
fwrite(fid, x, 'float');
fclose(fid);

%LPF_DESIGN Returns a discrete-time filter object.
% MATLAB Code
% Generated by MATLAB(R) 9.3 and Signal Processing Toolbox 7.5.
% Generated on: 14-Nov-2017 16:24:06
% Equiripple Lowpass filter designed using the FIRPM function.
% All frequency values are normalized to 1.
N     = 251;   % Order
Fpass = 0.31;  % Passband Frequency
Fstop = 0.35;  % Stopband Frequency
Wpass = 1;     % Passband Weight
Wstop = 1;     % Stopband Weight
dens  = 20;    % Density Factor

% Calculate the coefficients using the FIRPM function.
b  = firpm(N, [0 Fpass Fstop 1], [1 1 0 0], [Wpass Wstop], {dens});
```

```
lpf = dfilt.dffir(b);
lpf = lpf .* 3;

fid = fopen('lpf_scaled.bin', 'wb');
fwrite(fid, size(lpf, 2), 'int');
fwrite(fid, lpf, 'float');
fclose(fid);

fid = fopen('digInterp.bin');
ydig = fread(fid, 'float');
fclose(fid);

fid = fopen('polInterp.bin');
ypol = fread(fid, 'float');
fclose(fid);

newFs = round(Fs * 3 / 2);
audiowrite('digInterp.wav', ydig, newFs);
audiowrite('polInterp.wav', ypol, newFs);

n = [0:99999];
c16 = cos((2*pi/16).*n);
c8 = cos((2*pi/8).*n);
c4 = cos((2*pi/4).*n);

fid = fopen('c16.bin', 'wb');
fwrite(fid, size(c16, 2), 'int');
fwrite(fid, c16, 'float');
fclose(fid);

fid = fopen('c8.bin', 'wb');
fwrite(fid, size(c8, 2), 'int');
fwrite(fid, c8, 'float');
fclose(fid);

fid = fopen('c4.bin', 'wb');
fwrite(fid, size(c4, 2), 'int');
fwrite(fid, c4, 'float');
fclose(fid);

fid = fopen('polc16.bin');
c16pol = fread(fid, 'float');
fclose(fid);

fid = fopen('polc8.bin');
c8pol = fread(fid, 'float');
fclose(fid);

fid = fopen('polc4.bin');
c4pol = fread(fid, 'float');
fclose(fid);

fid = fopen('digc16.bin');
c16dig = fread(fid, 'float');
fclose(fid);

fid = fopen('digc8.bin');
c8dig = fread(fid, 'float');
fclose(fid);

fid = fopen('digc4.bin');
c4dig = fread(fid, 'float');
fclose(fid);
```